AD-A038 191 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
IMPLEMENTATION OF AN ARRAY BOUND CHECKER,(U)
NOV 76 N SUZUKI, K ISHIHATA F44620-73-C-0074
UNCLASSIFIED AFOSR-TR-77-0329 NL

| OF |

AD
A038191

END
DATE
FILMED
5-77

IMPLEMENTATION OF AN ARRAY BOUND CHECKER

Norihisa Suzuki

Kiyoshi Ishihata

November 1976

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

IMPLEMENTATION OF AN ARRAY BOUND CHECKER

by

Norihisa Suzuki
Department of Computer Science
Carnegie-Mellon University

and

Kiyoshi Ishihata
Department of Information Science
University of Tokyo
Tokyo,Japan

November 1976

This research was supported in part by the Defense Advanced Research Projects Agency (Contracts: F44620-73-C-0074, monitored by the Air Force Office of Scientific Research, and DAHC-~~15~~-72-C-0308) and in part by the University of Tokyo Computation Center. The views expressed are those of the authors.

UCL

See 1473

403 081

Abstract:

A practical system to check the correctness of array accesses automatically before actually running programs has been implemented. The system does not require any modification to input programs in the form of assertions or user interaction to guide proofs. That is, the system generates assertions to prove, synthesizes loop invariants, and finally proves verification conditions without interaction. A powerful proof strategy is invented which makes the time to check programs almost linear to the size of programs, yet the system can completely verify the correctness of array accesses of programs like tree sort and binary search with processing speed of about fifty lines per ten seconds. A three hundred line program example is also shown.

Keywords:

automatic program verification, semantic checker, array bound checker, induction iteration method, automatic synthesis of loop invariants, linear solver, theorem prover, frame problems.

1. INTRODUCTION

This paper describes a system which checks correctness of array accesses automatically without any inductive assertions or human interaction. For each array access in the program a condition that the subscript is greater than or equal to the lower bound and a condition that the subscript is smaller than or equal to the upper bound are checked and the results indicating within the bound, out of bound, or undetermined are produced. It can check ordinary programs at about fifty lines per ten seconds, and it shows linear time complexity behavior.

It has been long discussed whether program verification will ever become practical. The main argument against program verification is that it is very hard for a programmer to write assertions about programs. Even if he can supply enough assertions, he must have some knowledge about logic in order to prove the lemmas (or verification conditions) obtained from the verifier.

However, there are some assertions about programs which must always be true no matter what the programs do; and yet which cannot be checked for all cases. These assertions include: integer values do not overflow, array subscripts are within range, pointers do not fall off NIL, cells are not reclaimed if they are still pointed to, uninitialized variables are not used.

Since these conditions cannot be completely checked, many compilers produce dynamic checking code so that if the condition fails, then the program terminates with proper diagnostics. These dynamic checking code sometimes take up much computation time. It is better to have some checking so that unexpected overwriting of data will not occur, but it is still very awkward that the computation stops because of error. Moreover, these errors can be traced back to some other errors in the program. If we can find out whether these conditions will be met or not before actually running the program, we can benefit both by being able to generate efficient code and by being able to produce more reliable programs by careful examination of errors in the programs. Similar techniques can be used to detect semantically equivalent subexpressions or redundant statements to do more elaborate code movement optimization.

The system we have constructed runs fast enough to be used as a preprocessor of a compiler. The system first creates logical assertions immediately before array elements such that these assertions must be true whenever the control passes the assertion in order for the access to be valid. These assertions are proved using similar techniques as inductive assertion methods. If an array element lies inside a loop or after a loop a loop invariant is synthesized. A theorem prover was created which has the decision capabilities for a subset of arithmetic formulas. We can use this prover to prove some valid formulas, but we can also use it to generalize nonvalid formulas so that we can hypothesize more general loop invariants.

Theoretical considerations on automatic synthesis of loop invariants have been taken into account and a complete formula for loop invariants was obtained. We reduced the problem of loop invariant synthesis to the computation of this formula. This new approach of the synthesis of loop invariants will probably give more firmer basis for the automatic generation of loop invariants in general purpose verifiers.

2. THEORETICAL BASIS.

The correctness of array accesses can be stated within the theoretical framework of the weak correctness of programs. That is , we only have to show that the assertions placed immediately before the array element stating that the subscript expressions are within the defined bounds of the array hold, whenever control of the program comes to the assertions.

The major problem for making an automatic verifier which does not require any assertions by programmers is that the system must somehow invent loop invariants. Some research has been conducted toward automating the generation of loop invariants[3,4,10]. A common characteristic of all the research is that the method depends on heuristics. That is the system proposes some assertion as the loop invariant and let the prover decide if the program is provable from the loop invariant. The difficulty is that if it does not work, it is hard to see whether the program is not correct or the heuristics are wrong.

What we will do here instead is to obtain a complete formula for loop invariants. Just like Taylor's series expansion of functions will give a complete description of the function even though they are not usually calculable and infinite chain of approximations , we obtain an infinite chain of approximations to the general loop invariants from this formula.

Furthermore, if the assertion we want to prove is not a correct assertion we cannot invent a loop invariant which is true at entry to the loop and which is always true whenever control comes back to the top of the loop, and finally which implies the exit condition.

So, what we can hope to obtain is a formula which is a loop invariant if and only if the assertion is correct.

This formula is similar to the weakest precondition of Dijkstra [2]. What is different here is that we are only concerned about weak correctness. The formula is

$$\text{wlp(while C do S , Q)} = \forall i.i \geq 0 \supset W(i,C,S,Q)$$

where

$$W(0,C,S,Q) = \neg C \supset Q \text{ , and}$$
$$W(i+1,C,S,Q) = C \supset \text{wlp}(S,W(i,C,S,Q)) \text{ ,}$$
$$\text{for } i \geq 0.$$

Wlp stands for "weakest liberal precondition." The definitions of the weakest liberal precondition wlp(S , Q) is that if S is executed in the state satisfying wlp(S , Q) then Q is always true after termination of S, and no weaker condition satisfies such a condition. Wlp for assignment and conditional statements are the same as those of the weakest precondition.

It is easy to see that if wlp(while C do S , Q) is true at entry to the program then wlp(while C do S , Q) is always true whenever control comes back to the beginning of the loop. This is because

$$\text{wlp(while C do S , Q)} \wedge C \supset \text{wlp(S , wlp(while C do S , Q))}$$

It is easy to see that

$$\text{wlp(while C do S , Q)} \wedge \neg C \supset Q.$$

Also whenever the while statement terminates, Q is true at exit if and only if wlp(while C do S , Q) is true at entry.

Thus, this is the desired formula.

Note that no heuristics are involved in writing out the loop invariant. The problem is reduced to computing this formula, $\forall i.i \geq 0 \supset W(i,C,S,Q)$, and we can claim that $\forall i.j \geq i \geq 0 \supset W(i,C,S,Q)$ is the j-th approximation in a sense that it may be a loop invariant and if j-1 st approximation is a loop invariant then j-th approximation is certainly a loop invariant.

We will invent a procedure for checking whether j-th approximation is a loop invariant or not. Let L(j) stand for $\forall i. j \geq i \geq 0 \supset W(i,C,S,Q)$, the j-th approximation to the loop invariant. Certainly $L(j) \wedge \neg C \supset Q$. In order to establish that L(j) to be a loop invariant we have to show that L(j) is true at entry and also

$$L(j) \wedge C \supset wlp(S, L(j)).$$

But $wlp(S,L(j)) = wlp(S, \forall i.j \geq i \geq 0 \supset W(i,C,S,Q))$

$= \forall i.j \geq i \geq 0 \supset wlp(S,W(i,C,S,Q))$.

So $C \supset wlp(S, L(j))$

$= \forall i.j \geq i \geq 0 \supset W(i+1,C,S,Q)$.

That is

$$L(j) \wedge C \supset wlp(S,L(j))$$

is equivalent to

$$\forall i.j \geq i \geq 0 \supset (W(i,C,S,Q) \supset W(i+1,C,S,Q)).$$

So all we have to prove is to prove these two equations. There is a nice thing about this method and that is we can use all the results of computation up to j-1 st approximation to compute the j-th approximation. The reason is if W(i,C,S,Q) was failed to be proved then we can use this as an assumption for the next step and also we can back-substitute this formula around the loop and we can obtain W(i+1,C,S,Q).

This fact suggests an iterative method of proving weak correctness of programs without loop invariants. Because of this iterative nature we call it the " induction iteration method."

"Induction iteration method."

```
Step 1) Create $W(0,C,S,Q) = \neg C \supset Q$.

Step 2) Try to prove $W(i,C,S,Q)$ from
        $\forall k.i-1 \geq k \geq 0.W(k,C,S,Q)$. If it
        is true , the program is correct and
        the proof is done.

Step 3) We have to see if $W(i,C,S,Q)$ is true
        at entry to the loop. Back-substitute
        this $W(i,C,S,Q)$ through the program
        segment before the while statement.
        If it can be shown to be false at entry,
        the program is not correct and done.
        If it cannot be shown to be true, the
        algorithm halts indicating
        undetermined.

Step 4) We will use $W(i,C,S,Q)$ to prove the
        next step. So we will create
        $\forall k.i \geq k \geq 0 \supset W(k,C,S,Q)$. Then we
        will create $W(i+1,C,S,Q)$ from
        $W(i,C,S,Q)$ by the formula
        $C \supset wlp(S,W(i,C,S,Q))$.

Step 5) $i \leftarrow i+1$. Go to step 2).

end of algorithm
```

This iteration may never terminate. Particularly if the program is not correct we may very well not terminate. If we implement this algorithm, therefore, we have to put a bound on the number of iterations which determines the limitation of the system. We have to note here that the size of conditions , or the size of $W(i,C,S,Q)$, grows more rapidly than linear to the size of the program S. The reason, more than anything else, is that because of the rule

$$wlp(\text{if } B \text{ do } S1 \text{ else } S2,R) = (B \supset wlp(S1,R)) \wedge (\neg B \supset wlp(S2,R)),$$

the condition more than doubles in the size each time it is back-substituted through conditional statement. Since it is inevitable that the performance of a theorem prover is exponential to the size of formula, it is very important to keep the size of the condition W(i,C,S,Q) to be constant if we want to make a system works in linear time. For this reason we have developed a theorem prover which not only proves but also simplifies logical expressions, and modified the semantic rules wlp. These practical considerations will be discussed in the next two sections.

3. Theorem Prover

The synthesis of loop invariants is on a firmer ground, but we need to create a powerful theorem prover to make a practical system. The domain we are particularly interested in is an integer domain and formulas we have to prove is inequality relations with only universally quantified variables.

Before we prove these formulas all the arithmetic expressions and relations are converted to normal forms. Normal forms of arithmetic expressions and relations have been discussed in many verification literature[5].

As we have discussed in section 2, the main source of the exponential explosion in most of the verifiers comes from the growth of conditions to be proved. The theoretical limitations at least for the time being forbid us to create a theorem prover which behaves better than exponential time complexity. This suggests that instead of spending efforts in creating clever algorithms to reduce the speed of theorem prover by a constant factor, we should spend our efforts in creating simplification and generalization methods which limit the growth of conditions even though the size of the programs grow.

Since we are representing arithmetic expressions in normal forms, the size of expressions do not grow very rapidly by substitution of assignment statement. The problems are created by conditional statements. The detail of algorithms are discussed in the next section. In this section we will discuss about powerful theorem prover which are used to simplify conditions.

The basic algorithm of the theorem prover is King's linear solver [5], which is based on the Fourier-Motzkin method of linear programming. This class of prover is quite suited for array bound checking since array subscripts are in many cases linear expressions. The prover generally proceeds to show unsatisfiability of a set of linear inequalities.

Suppose x+e1 <= 0 , -x+e2 <= 0, 2*x+e3 <= 0 is the set we are going to show unsatisfiable, that is x+e1 <= 0 ∧ -x+e2 <= 0 ∧ 2*x+e3 <= 0 is false. The prover selects x to be the variable eliminated from the set. Then we classify this set into three subsets such that the coefficients of all the inequalities in the first set are positive, the coefficients of all the inequalities in the second set are negative, and each inequality in the third set does not contain x. We add each member of the first set and each member of the second set such that terms of x will disappear. We may have to multiply each inequality by some constant to adjust. If any one of them produces a contradictory formula the proof is successful and the process terminates. Otherwise we replace the original set by the union of the newly created set and the third subset of the original set. In this case e1+e2 <= 0, 2*e2+e3 <= 0 are the result of eliminating x. The procedure is iterated until we eliminate all the variables and obtain false statement, in which case the set is unsatisfiable, and otherwise satisfiable. Suppose the set is satisfiable and the result of elimination is a linear inequality e <= 0. Then - e + 1 <= 0 is the equation which is just sufficient to give unsatisfiability. -e + 1 <= 0 is in a sense the most general assumption. At this moment the system proposes -e+1<=0 to be the generalization of the lemma and tries to prove this instead. If there are several inequalities then each of them is in turn chosen to be the generalized lemma.

We will illustrate by an example how powerful these generalization techniques are.

```
VAR A: ARRAY[1:100] OF T;
  { 1 }
LOW ← 1 ;
  { 2 }
```

```
HIGH ← 100 ;
  { 3 }
WHILE LOW <= HIGH { 4 } DO
      BEGIN
      MIDDLE ← ( LOW + HIGH ) DIV 2 ;
        { 5 }
      IF A[MIDDLE] <= K
            THEN { 6 } HIGH ← MIDDLE-1
            ELSE { 7 } LOW ← MIDDLE+1
      END.
```

This is an essential part of a binary search algorithm which will be proved in section 5. One of the condition we have to show is here that 1 <= MIDDLE at { 5 }. Then what we first try to show is

```
W(0): LOW <= HIGH ⊃
        1 <= (LOW+HIGH) DIV 2
```

or after simplification

```
W(0): LOW <= HIGH ⊃ 2 <= LOW+HIGH
```

at { 4 }.

Since this is not provable, we try to see if this is true when control first enters the loop. We back-substitute W(0) through two statements and we obtain true at { 1 } . Now W(1) is formulated as

```
W(1): LOW <= HIGH ⊃
        (A[(LOW+HIGH) DIV 2] <= K ⊃
        (LOW <= ((LOW+HIGH) DIV 2 -1) ⊃
        2 <= LOW-1+(LOW+HIGH) DIV 2))∧
        (A[(LOW+HIGH) DIV 2] > K ⊃
        ((LOW+HIGH) DIV 2 +1 <= HIGH ⊃
        2 <= (LOW+HIGH) DIV 2 +1+HIGH))
or W(1): (LOW <= HIGH ∧
        A[(LOW+HIGH) DIV 2] <= K ∧
        LOW+2 <= HIGH ⊃
              6 <= 3*LOW+HIGH) ∧
        (LOW <= HIGH ∧
        A[(LOW+HIGH) DIV 2] > K ∧
        LOW+1 <= HIGH
              ⊃ 2 <= LOW+3*HIGH)
```

at { 4 } by back-substituting W(0) through the while body. W(0) ⊃ W(1) is

```
( 2<= LOW+HIGH ∧
LOW+2 <= HIGH ∧
A[(LOW+HIGH) DIV 2] <= K ⊃
        6 <= 3*LOW+HIGH) ∧
( 2 <= LOW+HIGH ∧
LOW+1 <= HIGH ∧
A[(LOW+HIGH) DIV 2] <= K ⊃
        2 <= LOW+3*HIGH )
```

The latter conjunct is valid, so W(0) ⊃ W(1) is simplified to

```
2 <= LOW+HIGH ∧
LOW+2 <= HIGH ∧
A[(LOW+HIGH) DIV 2] <= K ⊃
        6 <= 3*LOW+HIGH .
```

W(1) is easily shown to be true when control first enters the loop, but not itself provable. Therefore, we compute W(2) by back-substituting W(1) around the loop. That will not be provable again. This iteration many not terminate for some time. Now, let us look at the same example using the generalization by the prover. W(0) is not valid when it is computed. The computation is to show the unsatisfiability of

```
LOW <= HIGH   and   LOW+HIGH <= 1.
```

The elimination of HIGH generates

```
2*LOW-1 <= 0
```

and if we have -LOW+1 <= 0 , we can show the unsatisfiability . We set up -LOW+1 <= 0 as the generalization of W(0) to be proved. This is shown to be true when control first enters the loop. To see this is true when back-substituted around the loop we compute

```
wlp(S,-LOW+1 <= 0) =
    (A[MIDDLE] <= K ⊃
            -LOW+1 <= 0)∧
    (A[MIDDLE]>K ⊃
            -(LOW+HIGH) DIV 2 <= 0).
```

This is easily shown to be true from the assumptions

-LOW+1 <= 0 and LOW <= HIGH.

As you can see the generalization by the prover not only simplifies the problem but also generalizes the approximations of loop invariants to enable proofs in many cases.

4. SYSTEM CONFIGURATION

The system configuration is straight forward closely following the implementation suggested by the previous two sections. Only simplification not discussed is the treatment of conditional statements.

4. 1. Extraction of Local Conditions

A program is scanned once in the order they are presented as a text. Whenever an array element A[e] is found , conditions lower_bound_of (A) <= e and e <= upper_bound_of(A) are created as the conditions of the innermost statement containing the array element. We call these as bound assertions. If the statement is an assignment statement or a conditional statement , the condition must be true immediately before the statement. If the statement is a while statement, the condition must be true at entry and for each subsequent iteration of the loop.

4. 2. Semantic Treatment of Statements.

Each bound assertion created for an array access is transformed as it is back-substituted through the statements according to the semantic definitions of the statements. These assertions are at the same time simplified using normalization and theorem prover. The back-substitution process terminates either when there are no more statements in front, the condition is proved to be true or false, or it hits a while statement. Since we cannot iterate on a while statement indefinitely we return the result undetermined (U) if we fail to prove or disprove within a certain number of iterations.

The semantic definitions of statements are in principle the weakest liberal precondition rules, but we use some simplification as we transform conditions.

1) Statement lists.

If we have a statement list S1; S2 and suppose P is the assertion to prove after S2, then we obtain precondition of P over S2, wlp(S2,P). If this precondition is true or false or undetermined, we terminate the process and return the corresponding result. Otherwise we compute the precondition of wlp(S2,P) over S1 wlp(S1 , wlp(S2 , P)) and return that as the result.

2) Assignment statement.

If we have an assignment to a simple variable $X \leftarrow f(Y)$, then we return Substitute(f(Y) , X , P) . Here Substitute(e , X , P) is an expression obtained from P by substituting all the occurrences of X by e. If the statement is an assignment to an array element $A[e] \leftarrow f(Y)$, then Substitute(<A,e,f(Y)>,A,P) is returned, where <A,e,f(Y)> is an array obtained from A by substituting e-th element by f(Y).

3) If statement.

In the case of the if-then-else statement

if C then S1 else S2, we compute both the precondition of P over S1, wlp(S1,P) , and the precondition of P over S2, wlp(S2,P). If any one of them is false or undetermined, the process terminates immediately with the corresponding result as the result of back-substitution. If wlp(S1,P) and wlp(S2,P) are both true then return true. If wlp(S1,P) is true then we compute $\neg C \supset$ wlp(S2,P) by the theorem prover and return the generalized formula. If wlp(S2,P) is true then we compute $C \supset$ wlp(S1,P) by the theorem prover and return the generalized formula. Otherwise we compute ($C \supset$ wlp(S1,P)) $\wedge$ ($\neg C \supset$ wlp(S2,P)) by the theorem prover and return the conjunction of generalized formula.

Next, if the statement is if-then statement, namely, if it is of the form if C then S; then we compute precondition of P over the statement S , wlp(S , P). If wlp(S , P) is false or undetermined, we terminate the computation and return the wlp(S,P) as the value of back-substitution.

If wlp(S , P) is true, we return the generalized value of $\neg C \supset P$. Otherwise we return the generalized value of $(C \supset \mathrm{wlp}(S , P)) \wedge (\neg C \supset P)$.

4) While statement.

The semantic definition of a while statement while C do S relative to a post condition P can be defined in two cases depending on where P comes from. The first case is that P is created as the precondition of S. Then

$C \supset P$

is the first approximation of the loop invariant. If P is the precondition of the statement immediately after the while statement in a statement list, then

$\neg C \supset P$

is the first approximation of the loop invariant. Let these first approximations to be W(0). We compute W(0) and if W(0) is true then we return true as the result of the back-substitution. Otherwise we first back-substitute W(0) through the statements preceding the while statement. If the result is undetermined or false we return it as the result. Otherwise we create W(1) by the formula

$C \supset \mathrm{wlp}(S , W(0))$.

We generalize $W(0) \supset W(1)$ by the theorem prover and repeat the similar process.

4. 3. Creation of Assumptions.

Since there are a number of array accesses, we want to use the results of previous proofs so that we do not have to do similar deductions over and over again. For this purpose we have a mechanism for inserting assumptions in the program. All the array accesses are scanned in the order they appear in the text. As soon as the array bound assertions are proved they are inserted just in front of the statement in which the array access occurs. These assumptions are used for proofs of other array bound assertions. Since the rest of the bound assertions are always created after these assumptions, most of the assumptions are effectively used to prove or to simplify back-substituted assertions.

4. 4. Analysis of Loops.

For each while statement while C do S a set of variables of S which may be changed in the course of execution of S is collected. If we want to prove W(i) to be the invariant of this while statement, we first see if any of the variables of W(i) appear in this list of the changed variables. If they do not occur in this list, W(i) will not change by back-substitution and it is an invariant of the loop if W(i) is true at entry. We can save the computation to back-substitute in some cases.

4. 5. Preferential Elimination of Variables.

As we generalize and prove conditions, the choice of variables to eliminate is important for various resons. One strategy may be to try to choose a variable which has only one member in the set of positive coefficient set or in the negative coefficient set. This is because we do not want to explode the number of inequalities. In this system what we do is to choose variables which might be changed in the course of the execution of the while body. This strategy is useful for generalization because if we can eliminate variables which may be changed totally from the condition, then we can use the strategy described in 4.4. and we no longer have to back-substitute the while body to determine the invariance of the condition.

5. EXAMPLES

The following binary search program has been proved that all the array accesses are within bounds. The proof required 2 CPU seconds.

```
TYPE TABLE=ARRAY[1:100] OF INTEGER;
PROCEDURE BINARYSEARCH(VAR A:TABLE
   ;KEY:INTEGER;VAR MIDDLE:INTEGER);

VAR LOW,HIGH: INTEGER;

BEGIN
 { 1 }
HIGH := 100;
 { 2 }
LOW := 1;
WHILE LOW ≤ HIGH { 3 }  DO
      BEGIN
      MIDDLE := (LOW+HIGH) DIV 2;
       { 4 }
      IF A[MIDDLE]=KEY
           THEN { 5 }  LOW:=HIGH+1
      ELSE IF A[MIDDLE] > KEY
           THEN { 6 } HIGH := MIDDLE-1
      ELSE { 7 }  LOW := MIDDLE+1
      END;
       { 8 }
IF A[MIDDLE] ≠ KEY THEN MIDDLE := 0
END;
```

There are three accesses of A, two within the loop and one outside of the loop. We are going to show how the system proves that the array accesses are within the ranges. The proofs are not trivial, since MIDDLE does not change monotonically. Also there is an integer division and we have to make sure that the right truncation is performed when we normalize expressions. We also have to prove that MIDDLE keeps within range even after the end of the execution of while statement. The system searches an array element textually from the beginning of the program. It first finds A[MIDDLE] in the statement IF A[MIDDLE]=KEY THEN ... . It creates an assertion

1 - MIDDLE <= 0

at location { 4 } to insure the correctness of the array access relative to the lower bound of the array A. This assertion is back-substituted and becomes

1 - (HIGH + LOW) DIV 2 <= 0

at location { 3 }. This is normalized to

2 - HIGH - LOW <= 0.

Since the loop condition (condition of the while statement) is

-HIGH + LOW <= 0,

-HIGH + LOW <= 0 ⊃ 2 -HIGH -LOW <= 0

is proposed as the first approximation of the loop invariant. The theorem prover generalizes it to

1 - LOW <= 0.

To show that this condition is true when control first enters the while statement this condition is back-substituted to { 2 }. At location { 2 } it becomes

1 - 1 <= 0

and it is proved. For the proof of the inductive hypothesis,

1 - LOW <= 0

is assumed to be true at { 3 } . This is back-substituted through the loop body. Since there is a three way branch, three subproblems are created corresponding to each branch. A path through branch { 5 } creates

A[MIDDLE]=KEY ⊃ -HIGH <= 0 .

This has to be proved by the inductive hypothesis

1 - LOW <= 0

and the loop condition

- HIGH + LOW <= 0

at loop entry { 3 }. These two assumptions imply

1 - HIGH <= 0

and clearly

- HIGH <= 0.

The second subgoal is created by the path through location { 6 } and, since no modification is done, it is clearly an invariant through this path. The final subgoal is created by the path through { 7 }. It is

- MIDDLE <= 0.

at { 4 } and

- LOW - HIGH <= 0

at { 3 }. This is again proved from the inductive hypothesis and the loop invariant.

To prove that MIDDLE is smaller than the upper bound of the array A,

MIDDLE <= 100

is created and the first approximation to the loop invariant at { 3 } becomes

HIGH - 100 <= 0,

using the generalization.

The three subgoals are

1) HIGH - 100 <= 0
2) HIGH + LOW <= 203
3) HIGH - 100 <= 0.

1) and 3) are the same as by the inductive hypothesis. 2) can be shown to be true by the inductive hypothesis and the loop invariant.

At this point two assumptions, 1 - MIDDLE <= 0 and -100 + MIDDLE <= 0 , are created and stored at { 4 }.

The second array access is proved to be within the bounds using these assumptions.

The third array access is immediately after the loop. All we have to do is to show that

$\neg$ (-HIGH+LOW <= 0) $\supset$ 0 <= MIDDLE <= 100

is true at the loop head every time control passes this location. This can be proved similarly.

The following is the output from the system. The structure of the program is essentially maintained, and the array elements have modified outputs indicating the results of checking. The format is

```
<array name> [ <check result> $
        <subscript expression> $ <check result> ]
```

where <check result> is any one of I, O, U. I means the subscript is within range, O means the subscript is out of range, and U means that the system cannot determine either way. Typical output looks like

```
A[I$e$U] + B[I$e+f$O]
```

which means e is greater than or equal to the lower bound of A, but it is not clear whether e is less than or equal to the upper bound of A. e+f is greater than or equal to the lower bound of B, but it is greater than the upper bound of B.

```
*****
TYPE TABLE=ARRAY[1:100] OF INTEGER;

PROCEDURE BINARYSEARCH(VAR A:TABLE
     ;KEY:INTEGER;VAR MIDDLE:INTEGER);
  VAR LOW,HIGH:INTEGER;
  BEGIN
     HIGH := 100;
     LOW := 1;
     WHILE -HIGH+LOW<= 0 DO
        BEGIN
          MIDDLE := HIGH+LOW DIV 2;
          IF A[I$MIDDLE$I]=KEY
             THEN LOW := 1+HIGH
```

```
            ELSE
            IF 1+KEY-A[I$MIDDLE$I]<=0
               THEN HIGH := -1+MIDDLE
                  ELSE LOW := 1+MIDDLE
         END;
      IF ¬A[I$MIDDLE$I]=KEY
         THEN MIDDLE := 0
   END;
```

*****

TIME: 2 CPU SECS

The next example is the tree sort. This is to show that some of the more difficult arithmetic operations like multiplication by a constant can be handled properly. It is not very difficult for a person to observe that all the array accesses by the subscript J are done correctly, since in the inner most loop J is increasing monotonically and the loop condition is J <= N. However, among the array accesses with subscript I the second and the third array accesses are not trivial. Even for a human it needs a clear understanding of how this program works and how I and J are used. Once we know that J = 2*I at the loop head, we know that since J is monotonically increasing so is I. I <= 100 is maintained because at the loop head the value of I is either the same as the value of J (or 1 greater than J if J < N and A[J] < A[J+1]) of the previous iteration and at that time J is less than or equal to N. This is informal human reasoning. The array bound checker does not use similar reasoning, but it manages to prove this by systematic inductive iteration and generalization by the prover. Another thing that needs mentioning is that this program was checked with 16 CPU seconds, which is within the usable range, especially considering that the system is written in LISP 1.6.

*****

```
VAR A:ARRAY[1:100] OF T;
   K,I,J,N:INTEGER;
   COPY,WORK:T;
BEGIN
   K := 100 DIV 2;
   WHILE 2-K<= 0 DO
      BEGIN
         I := K;
         N := 100;
         COPY := A[I$I$I];
         J := 2*I;
         WHILE J-N<= 0 DO
            BEGIN
               IF 1+J-N<= 0
                   THEN IF 1+A[I$J$I]-A[I$1+J$I]<=0
                        THEN J := 1+J;
               IF 1+COPY-A[I$J$I]<= 0
                   THEN BEGIN
                       A[I$I$I] := A[I$J$I];
                       I := J;
                       J := 2*I
                     END ELSE J := N+1
            END;
         A[I$I$I] := COPY;
         K := -1+K
      END;
    K := 100;
    WHILE 2-K<= 0 DO
      BEGIN
         I := 1;
         N := K;
         COPY := A[I$I$I];
         J := 2*I;
         WHILE J-N<= 0 DO
            BEGIN
               IF 1+J-N<= 0
                  THEN IF 1+A[I$J$I]-A[I$1+J$I]<=0
                        THEN J := 1+J;
               IF 1+COPY-A[I$J$I]<= 0
                   THEN BEGIN
                        A[I$I$I] := A[I$J$I];
                        I := J;
                        J := 2*I
                     END ELSE J := N+1
            END;
         A[I$I$I] := COPY;
         WORK := A[I$1$I];
         A[I$1$I] := A[I$K$I];
         A[I$K$I] := WORK;
         K := -1+K
    END
END
*****
```

TIME: 16 CPU SECS

This next example is almost identical to the previous program except the conditional statement has now been changed from

```
IF J<N THEN IF A[J]<A[J+1] THEN J := J+1     to
IF J<N ∧ A[J]<A[J+1] THEN J := J+1.
```

PASCAL evaluates logical expression in parallel so J+1 may become greater than N. Thus, the error has been correctly detected and this is a valuable information to the programmer.

*****

```
VAR A:ARRAY[1:100] OF T;
   K,I,J,N:INTEGER;
   COPY,WORK:T;
BEGIN
   K := 100 DIV 2;
   WHILE 2-K<= 0 DO
      BEGIN
         I := K;
         N := 100;
         COPY := A[I$I$I];
         J := 2*I;
         WHILE J-N<= 0 DO
            BEGIN
               IF 1+J-N<= 0 ∧ 1+A[I$J$I]-A[I$1+J$U]<= 0
                  THEN J := 1+J;
               IF 1+COPY-A[I$J$I]<= 0
                  THEN BEGIN
                        A[I$I$I] := A[I$J$I];
                        I := J;
                        J := 2*I
                     END ELSE J := N+1
            END;
         A[I$I$I] := COPY;
         K := -1+K
      END;
   K := 100;
   WHILE 2-K<= 0 DO
   BEGIN
     I := 1;
     N := K;
     COPY := A[I$1$I];
     J := 2*I;
     WHILE J-N<= 0 DO
       BEGIN
         IF 1+J-N<= 0 ∧ 1+A[I$J$I]-A[I$1+J$U]<= 0
            THEN J := 1+J;
         IF 1+COPY-A[I$J$I]<= 0
            THEN BEGIN
                  A[I$I$I] := A[I$J$I];
                  I := J;
                  J := 2*I
                END ELSE J := N+1
       END;
     A[I$I$I] := COPY;
     WORK := A[I$1$I];
     A[I$1$I] := A[I$K$I];
     A[I$K$I] := WORK;
     K := -1+K
   END
END
*****
```

TIME: 20 CPU SECS

This final example is taken from lexical analyzer of PASCAL compiler. The program has been modified from the original program by taking out several repeat statements, so it is not a correct program from the standpoint of lexical analyzer correctness. However, we have been able to put this program through the checker without any intermediate assistance. The input is 300 lines of code including some comments. It takes 45 seconds to process. The system actually found an error at location { 1 }. Whenever buflen exceeds 256, the system prints out error messages, but it does not reset buflen. Therefore, the following array element BUF[buflen] will be out of bound. The system cannot check the array access at location { 2 } , because bufindex is a global variable. This is the kind of examples we need some assistance from the programmer stating the behavior of global variables and parameters.

```
TYPE STnode=RECORD ctindex:INTEGER;
      length:subnamelen END;

TYPE Treenode=RECORD ctindex:INTEGER;
      stindex:INTEGER;
      left:↑Treenode;
      right:↑Treenode END;

TYPE Tree=↑Treenode;

TYPE word=integer;

TYPE CTnode=word;

TYPE pwords=ARRAY[1:3] OF word;

VAR ST:ARRAY[1:50] OF STnode;
  CT:ARRAY[1:50] OF CTnode;
  Toptree:Tree;
  P:Tree;
  curname:ARRAY[1:15] OF CHAR;
  curlen:subnamelen;
  CH:CHAR;
  BUF:ARRAY[1:255] OF CHAR;
  LETTERS,LETTERSORDIGITS:SETOFCHAR;
  first,eofinput:BOOLEAN;
  i:subnamelen;
  a:pwords;
FUNCTION ROUNDUP(CONST num:integer;denom:integer)
      :integer;
  VAR q:integer;
  BEGIN
    q := num DIV denom;
    IF num MOD denom=0
       THEN ROUNDUP := q
      ELSE ROUNDUP := 1+q;
  END;

PROCEDURE COPYLINE(CONST DUMMY:ghost);
  VAR c:CHAR;
  BEGIN
    IF NOT(EOF(INPUT))
       THEN BEGIN
         buflen := 0;
         WHILE NOT(EOLN(INPUT)) DO
           BEGIN
             READ(c);
```

```
            buflen := 1+buflen;
            IF 256-buflen<= 0
               THEN WRITELN(TTY , ERROR1);
              { 1 }
            BUF[I$buflen$U] := c;
          END;
        READLN(DUMMY);
        WRITELN(TTY);
        bufindex := 0;
      END
    ELSE eofinput := TRUE;
  END;

PROCEDURE NEXTCH(CONST DUMMY:ghost);
  BEGIN
    bufindex := 1+bufindex;
    IF 1-bufindex+buflen<= 0
       THEN IF bufindex-buflen=1
          THEN CH := BLANK
          ELSE BEGIN
              COPYLINE(DUMMY);
              bufindex := 1;
              CH := BUF[I$1$I];
            END
            { 2 }
      ELSE CH := BUF[U$bufindex$U];
  END;

PROCEDURE GETNAME(CONST DUMMY:ghost);
  BEGIN
    curlen := 1;
    curname[I$1$I] := CH;
    NEXTCH(DUMMY);
    WHILE INSET(CH,LETTERSORDIGITS) DO
      BEGIN
        IF -14+curlen<= 0
           THEN BEGIN
              curlen := 1+curlen;
              curname[I$curlen$I] := CH
            END;
        NEXTCH(DUMMY);
      END;
  END;

PROCEDURE PACKSTRING(VAR a:pwords; VAR len:subnamelen);
  VAR k,shift:integer;
  BEGIN
    i := 0;
  wordindex := 0;
  shift := 0;
  WHILE 1+i-curlen<= 0 DO
    BEGIN
      IF shift=0
        THEN BEGIN
          wordindex := 1+wordindex;
          a[I$wordindex$U] := 0;
          shift := 1;
         END;
      i := 1+i;
      IF shift=B10000000000
        THEN BEGIN
          k := -B1000000000*B40+
            B1000000000*ORD(curname[I$i$U]);
          k := 2*k;
         END
       ELSE k := -B40*shift+
            shift*ORD(curname[I$i$U]);
      a[U$wordindex$U] := k+a[U$wordindex$U];
      shift := B100*shift;
    END;
  len := curlen;
 END;

PROCEDURE FINDPLACE(VAR a:pwords;len:subnamelen;
             VAR first:boolean;Top:Tree;VAR pintotree:Tree);
  VAR P:Tree;
    L:subnamelen;
    ctlessa,ctless0,eq:boolean;
    wordind:integer;
  BEGIN
    IF Top=NIL
      THEN BEGIN
        first := TRUE;
       END
     ELSE BEGIN
        BEGIN
          L := ST[U$Top↑.stindex$U].length;
          IF L=len
            THEN BEGIN
              eq := TRUE;
              k := ROUNDUP(L,6);
              wordind := 1;
              WHILE wordind-k<= 0∧eq=TRUE DO
                BEGIN
                  IF ¬CT[U$-1+wordind+
                    Top↑.ctindex$U]=
              a[I$wordind$U]
               THEN eq := FALSE
              ELSE wordind := 1+wordind;
          END;
        IF eq=TRUE
           THEN BEGIN
              first := FALSE;
              pintotree := Top;
            END
          ELSE BEGIN
              ctlessa:=1-a[U$wordind$U]+
                 CT[U$-1+wordind+
                 Top↑.ctindex$U]<= 0;
              ctless0 := 1+CT[U$-1+
              wordind+Top↑.ctindex$U]<=0;
              IF ¬ctless0=1+
                a[U$wordind$U]<= 0
                 THEN ctlessa := ctless0;
              IF ctlessa
                 THEN BEGIN
                    P := Top↑.right;
                    FINDPLACE(a,len,
                      first,P,pintotree);
                    IF Top↑.right=NIL
                       THEN Top↑.right:=
                        pintotree;
                  END
                ELSE BEGIN
                    p := Top↑.left;
                    FINDPLACE(a,len,
                      first,P,pintotree);
                    IF Top↑.left=NIL
                       THEN Top↑.left
                         :=pintotree;
                  END;
            END;
      END
    ELSE IF 1-len+L<= 0
         THEN BEGIN
           P := Top↑.right;
           FINDPLACE(a,len,first,
             P,pintotree);
           IF Top↑.right=NIL
              THEN Top↑.right:=
               pintotree;
         END
       ELSE BEGIN
           P := Top↑.left;
```

```
                   FINDPLACE(a,len,first,P,
                     pintotree);
                   IF Top↑.left=NIL
                      THEN Top↑.left:=pintotree;
                 END;
         END;
       END;
  END;

PROCEDURE INITNODES(VAR a:pwords;len:subnamelen;pnode:Tree);
  BEGIN
    BEGIN
      IF 51-availCT<= 0
         THEN WRITELN(TTY , ERROR2);
      IF 51-availST<= 0
         THEN WRITELN(TTY , ERROR3);
      pnode↑.ctindex := availCT;
      pnode↑.stindex := availST;
      pnode↑.left := NIL;
      pnode↑.right := NIL;
    END;
    k := ROUNDUP(len,6);
    BEGIN
      ST[U$availST$U].length := len;
      ST[U$availST$U].ctindex := availCT;
    END;
    availST := 1+availST;
    availCT := availCT+k;
  END;

BEGIN
  availST := 1;
  availCT := 1;
  Toptree := NIL;
  eofinput := FALSE;
  WRITELN(TTY , TITLE);
  WRITELN(TTY);
  COPYLINE(DUMMY);
  GETNAME(DUMMY);
  WHILE NOT(eofinput) DO
    BEGIN
      PACKSTRING(a , curlen);
      FINDPLACE(a , curlen , first , Toptree , P);
      IF Toptree=NIL
         THEN Toptree := P;
      IF first
         THEN BEGIN
            INITNODES(a , curlen , P);
            WRITE(TTY , FIRSTUSE);
          END
        ELSE WRITE(TTY , REPEATED);
      WRITELN(TTY);
      GETNAME(DUMMY)
    END;
END
*****

TIME: 45 CPU SECS
```

6. CONCLUSION.

There are several limitations to what the system can do.

One class of problems which this system cannot do is to check the correctness of array accesses in a loop if the correctness depends on some data whose values are set before the execution of the loop. One good example is a very widely used class of techniques to speed up the sequential search by storing some exceptional data at the end of the array so that the comparison loop always terminates. The program is

```
VAR A:ARRAY [1:100] OF T;
BEGIN
A[100] ← KEY;
I ← 1;
WHILE A[I] > KEY DO
      I ← I+1;
END.
```

The while loop terminates because A[100] is the same as KEY and that will make the while condition false. Using our methods we will not be able to prove this, since what we have to prove as the induction step is

$$I <= 100 \wedge A[I] > KEY \supset I <= 99.$$

Conventional program verifiers using Floyd-Hoare logic system have the similar problem. That is even though a loop does not modify some portion of the data we have to declare these properties as the loop invariant. The first author noticed this problem [7] and has generally called it a " frame problem of inductive assertion method ", borrowing terminology from similar problems in artificial intelligence [6]. The solution to this problem is to extend the rules for loops so that properties of data can influence the proofs inside the while statements and after the while statements. Using Hoare's notation the new rule is

$$\frac{\begin{array}{l} P(x;v) \supset I(x;v), \\ P(x;v) \wedge I(x;v') \wedge C(x;v') \{\ S(x;v')\ \}\ I(x;v'), \\ P(x;v) \wedge I(x;v') \wedge \neg C(x;v') \supset Q(x;v') \end{array}}{P(x;v)\ \{\ \text{while } C(x;v) \text{ do } S(x;v)\ \}\ Q(x;v)}$$

where x is a set of variables which do not change their values within S(x;v) , v is a set of variables which may change their values within S(x;v) , and v' is a set of totally new variables corresponding to v. The correctness and implementation of this and other frame problem free rules for various syntax constructs are discussed in [7] and [8].

We can use this technique to prove the previously unsuccessful problem of linear search. We failed to prove

$$I <= 100 \wedge A[I] > KEY \supset I <= 99$$

at the loop head, but if we replace I by I' and back-substitute further to the front, we obtain

$$I'<=100 \wedge <A,100,KEY>[I'] > KEY \supset I'<=99$$

and which is now a valid statement for which we can show the correctness. Here <A,100,KEY> denotes an array obtained from A by replacing A[100] by KEY.

The second problem is that the generalization capabilities of the system are not good enough for many problems. Ultimately we must ask the user to give some assertions or on failure the system ought to indicate the reason for the failure and ask the guidance of the user.

We can say about the same thing for procedure and function calls and their definitions. When we are checking procedure definitions, conditions are back-substituted in the program just as we treat main programs. However, if we come to the beginning of the procedure body we can do two things. Either we say that the conditions cannot be determined and report to the programmer the reason of failure , or try to check that these conditions are true at each calling occurrence. We can also use induction iteration method to synthesize entry conditions for recursive programs. Currently we take the former approach but eventually we either have to ask programmers to put in a modest number of entry and exit assertions and also try to check all calling occurrences to see if the conditions are right.

The development and the availability of this kind of system will change programming language design just as verification and formal semantics have created a great effects on the programming language design. We think we will see a great number of features which enhance security and reliability will be put into programming systems, so that these properties can be checked at compile time and create more reliable programs without losing the run-time efficiency .

Because the time complexity is good and the size of a program it can handle is quite substantial, we believe that the production version of this system soon will be available to the public and verification will be directly benefitting the computing community.

ACKNOWLEDGEMENT

The first author is thankful for the discussions with Eiiti Goto, Paul Hilfinger, Jim Morris, Ben Wegbreit, and Bill Wulf.

REFERENCES.

[1] Cousot, P. & R. Cousot,
Static verification of dynamic type properties of variables,
Research Report # 25, Laboratoire d'Informatique,U.S.M.G.,Grenoble.

[2] Dijkstra, E. W.,
Guarded commands, nondeterminacy and formal derivation of programs,
Comm. ACM 18, 8, August, 1975, pp. 453-457.

[3] German, S.M. & B. Wegbreit,
A synthesizer of inductive assertions,
IEEE Trans. of Software Engineering, Vol. SE-1, No.1, March 1975, pp.68-75.

[4] Katz, S.M. & Z. Manna,
A logical analysis of programs,
Comm. ACM 19, 4, April, 1976, pp. 188-206.

[5] King, J.C.
A program verifier,
Ph.D. thesis, Dept. of Comp. Sci.,
Carnegie-Mellon University, September 1969.

[6] McCarthy, J. & P. J. Hayes,
Some philosophical problems from the standpoint of artificial intelligence,
Machine Intelligence 4, American-Elsevier , pp. 463-502.

[7] Suzuki, N.,
Automatic verification of programs with complex data structures,
Ph.D. thesis, Dept. of Comp. Sci., Stanford University,
STAN-CS-76-552, February, 1976.

[8] Suzuki, N.,
Iteration induction method,
In preparation.

[9] Suzuki, N.,
Frame problems in Floyd-Hoare logic,
In preparation.

[10] Wegbreit, B.,
The synthesis of loop predicates,
Comm. ACM 17, 2, Feb. 1974, pp. 102-112.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 19 / 18 AFOSR - TR - 77 - 0329 — 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 IMPLEMENTATION OF AN ARRAY BOUND CHECKER	5. TYPE OF REPORT & PERIOD COVERED Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Norihisa Suzuki, Kiyoshi Ishihata	8. CONTRACT OR GRANT NUMBER(s) 15 F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Sciencific Research (NM) Bolling AFB, DC 20332	12. REPORT DATE 11 Nov[redacted] 1976
	13. NUMBER OF PAGES 36 12 38 p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) 16 2304 17 A2	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

403081

20. ABSTRACT (Continue on reverse side if necessary and identify by block number) A practical system to check the correctness of array accesses automatically before actuallly running programs has been implemented. The system does not require any modification to input programs in the form of assertions or user interaction to guide proofs. That is, the system generates assertions to prove, synthesizes loop invariants, and finally proves verification conditions without interation. A powerful proof strategy is invented which makes the time to check programs almost linear to the size of programs, yet the system can completely verify the correctness of array accesses of programs like tree sort and binary search with processing speed of about fifty lines per ten seconds. A three hundred line program example is also shown.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)